WRITE ONCE.
SCALE ANYWHERE.

Scaling Spring Applications in 4 Easy Steps

Using GigaSpaces XAP and OpenSpaces

Nati Shalom, CTO | March 2008

Abstract

This paper is a brief guide to scaling Spring-based applications. It shows how to solve well-known problems that crop up when applications begin to scale out across multiple physical machines—a bottleneck in the data tier, a bottleneck in the messaging tier, a bottleneck caused by the tight coupling between business logic, data, and messaging, and a bottleneck caused by the limited methods that exist today to deploy and provision applications across multiple computers. We suggest a number of simple yet innovative steps, which leverage the idea of virtualization to help you release these bottlenecks, but do not require that you change your business logic code or otherwise re-work your application. The steps can be performed individually, as a targeted "cure" for each bottleneck, but together they form a holistic solution that leverages the Space-Based Architecture (SBA) to enable true linear scalability for your application.

Table of Contents

1. Introduction

This paper is a brief guide to scaling Spring-based applications, or more accurately, to dealing with bottlenecks that inevitably arise when you try to scale out contemporary applications across multiple physical machines. These include a bottleneck in the data tier, a bottleneck in the messaging tier, and a bottleneck caused by the limited methods that exist today to deploy and provision applications across multiple computers. None of these are special to Spring-based applications, but Spring creates a special opportunity to get rid of them transaprently, just by making changes to your Spring configuration, with no impact on your application's business logic code.

GigaSpaces eXtreme Application Platform (XAP) takes advantage of this opportunity, allowing you to wire advanced middleware functionality directly into your Spring Application Context, without your application having to be aware of it. This is made possible by OpenSpaces, a Spring-based open source framework developed by GigaSpaces.[1]

In this paper we show how GigaSpaces XAP can offer targeted solutions to each of the bottlenecks listed above, by **virtualizing** different parts of your application and its middleware. The following table summarizes the bottlenecks and the relevant virtualization solution GigaSpaces provides.

Scalability Bottleneck	How Virtualization Can Help
Data—the central database is in the critical path of the business transaction. I/O delays slow the application down, and the database must be able to respond to increasing loads in real-time.	▪ Asynchronous persistence—I/O delay does not affect the application ▪ Data stored reliably in an in-memory cluster until it is persisted ▪ Writing to database in batches, from a collocated service, for highly optimized data access
Messaging—the central messaging provider is in the critical path of the transaction and might become overloaded. It might also impose an I/O delay, if the messaging provider uses persistence for high availability, as is often the case.	▪ Data and messaging in the same place, with one clustering model, guaranteeing affinity with no overhead ▪ No network hops between messaging and database, less moving parts ▪ High availability without disk access, using in-memory clustering ▪ Messaging can be done over a WAN
Business Logic—business logic must be tightly coupled to the data tier and messaging provider, to ensure data affinity and consistency.	▪ Data and messaging collocated with business logic services ▪ Application becomes a self-sufficient resource that can be deployed anywhere and scaled linearly ▪ Minimal possible end-to-end latency ▪ No moving parts, easier maintenance and administration
Deployment—a scaled-out application requires provisioning, maintenance and healing in case of failure. Doing all of this manually is expensive and cannot respond fast enough to failure.	▪ Proactive automatic deployment ▪ Provisioning sensitive to SLA and computing resources ▪ Self-healing—automatic detection and recovery from failure ▪ Aggregate monitoring with a holistic end-to-end view of business transactions

[1] To learn more about OpenSpaces, refer to *GigaSpaces Online Help*, OpenSpaces, http://www.gigaspaces.com/wiki/x/FIBI. The open source community website is at http://www.openspaces.org.

The remainder of this paper explains how to carry out four simple steps to achieve the results in the right-hand column, as a "cure" for each of the scalability bottlenecks. You can view these steps as targeted optimizations—in fact, you only need to perform the steps that target your specific problems. But they are actually much more than optimizations, because they set the stage for a holistic solution to the problems of scaling out stateful applications. Put together, these steps are a full roadmap towards adopting the Space-Based Architecture (SBA) and achieving linear scalability.[2]

To give you a taste of this, we end the paper with a real-life example of a four-week transition from Tier-Based Architecture to Space-Based Architecture, which involved nothing more than the four simple steps outlined in this paper.

[2] To learn more about SBA, a holistic architectural model which powers these virtualization steps, refer to the GigaSpaces white paper, "The Scalability Revolution: From Dead End to Open Road".

2. STEP ONE: Virtualizing Data

In this step, you will learn how your application can stop interacting directly with a database, and instead rely on an In-Memory Data Grid (IMDG)—a distributed network of in-memory data nodes—to take care of consistency, availability, and reliability of application state, while persisting data to disk in the background. This eliminates the I/O bottleneck, immediately improves latency, and isolates the database from your application's load.

Transition: **Virtualize the data tier using the *In-Memory Data Grid (IMDG)*.**

Result: **No I/O bottleneck, database isolated from application load.**

2.1. Persistence as a Service

When the IMDG is the 'system of record' for your application, persisting data to disk is simply a background process that can be handled on the back-end. The implication is that persistence to disk can be a service that is completely decoupled from the application.

In practical terms, this means that the application needn't be aware of anything related to data persistence: the store implementation, the method (database, log file, etc.), and the schedule are determined outside the context of the application, and can be changed at any time, even at runtime, without affecting the application's code or configuration. A persistent service is simply run on the network, and the IMDG takes care of synchronizing between the application and this abstracted service.

The application can request database storage for data that needs long-term persistence, and memory-based persistence for data of a more transient nature. The application can also specify when it is acceptable to persist data asynchronously. This still provides synchronous database persistence when needed, but when data doesn't need to be on disk or doesn't need to block the operation until it is persisted (which is often the case), the application can achieve a big performance boost without compromising on resiliency.

Existing applications and databases can be transparently migrated to this model, because the IMDG can persist data to a database using an O/R mapping solution like Hibernate.

2.2. Using Hibernate Second-Level Cache

Hibernate features such as Second-Level Cache provide basic integration designed for read-mostly scenarios in which developers can use the Hibernate API to abstract the interaction with the database. Using Hibernate to delegate a query into an In-Memory Cache saves the I/O overhead associated with retrieval of data that was already loaded prior to the query.

The benefit of this approach is that it can be made transparent to Hibernate users. The downside is that it is limited to read-mostly scenarios, it doesn't address the scalability challenge, and it still requires tight coupling between the application and the database.

You can overcome these limitations quite easily by using the IMDG as a front-end to Hibernate, which will give you a powerful distributed cache with multiple clients and advanced clustering. Instead of interacting directly with Hibernate, the application interacts with the IMDG. The IMDG handles the synchronization with the persistence service, which in turn uses a Hibernate plug-in to map the object view and the relational view in the background. *SQLQuery* remains the primary API

when accessing the data.[3]

2.3. Database Becomes In-Memory Data Grid

Defining GigaSpaces XAP as your Hibernate Second-Level Cache has its limitations. To get all the advantages of data virtualization, you'll need to go one step further and replace your database with the IMDG, leaving the rest of your application exactly as-is. This solves the I/O bottleneck and sets the stage for total virtualization of your application.

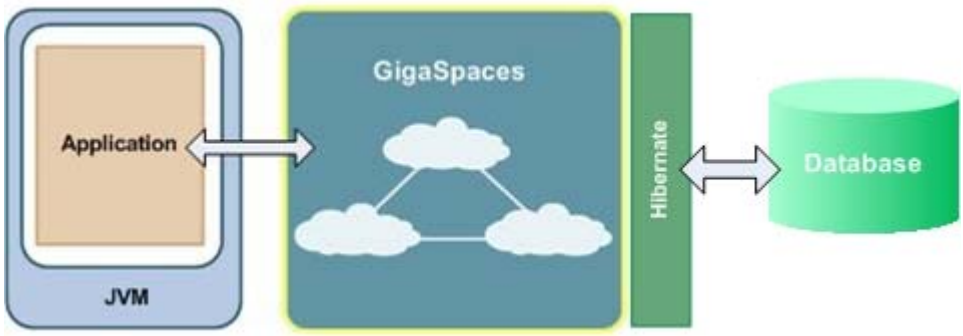


Figure 7 – Application interacts with GigaSpaces IMDG, with database in the background

In most cases the IMDG will be **partitioned**, with a backup node for each partition to provide high-availability. But many other topologies are also available, including replicated and master-local.[4] Because OpenSpaces uses the Persistence as a Service model, you can **tailor the persistence to suit your needs**:

- **The GigaSpaces Mirror Service** can perform transparent, asynchronous persistency of all IMDG data to the database.[5]
- The IMDG can also persist some or all data to a **log file** or any other persistent resource.
- It is possible to **write-through** certain types of data to a database of your choice, or **read-through** from a database when certain operations are performed on the IMDG.[6]
- Some types of data won't need persistence at all, given that they are kept highly available by means of replication between IMDG nodes.

[3] See *GigaSpaces Online Help*, Enterprise Data Grid Tutorial B - Aggregate Queries, http://www.gigaspaces.com/wiki/display/GS6/Enterprise+Data+Grid+Tutorial+B+-+Aggregate+Queries.

[4] See *GigaSpaces Online Help*, Data Grid Topologies, http://www.gigaspaces.com/wiki/x/c4GY.

[5] See *GigaSpaces Online Help*, Mirror Service, http://www.gigaspaces.com/wiki/x/7IGY.

[6] See *GigaSpaces Online Help*, Read-Through and Write-Through, http://www.gigaspaces.com/wiki/x/K4KY.

More Reliable than a Database

The IMDG, which uses the PaaS paradigm to persist data to disk in the background, is not only more efficient but also more reliable than a database:

- If the database crashes, the system continues to work and recovers automatically as soon as the database is up and running.
- It provides true hot fail-over within a few miliseconds, or a maximum of 2-3 seconds in a worst-case scenario such as a network cable coming unplugged. Fail-over happens on the client side, which enables seamless transtion between primary and backup.
- It monitors memory usage and will block operations before running out of memory, thus enabling graceful recovery and tracing of the cause of the event.

2.4. The Transition

To make things more concrete, we'll show how the transition to Step One is achieved in a simple reference application.

The Reference Application

Here is a description of our reference application:

- Designed according to the Tier-based Architecture.
- A large number of clients—Feeders and Receivers—manage their workflow using a JMS queue. For the purposes of the example, the JMS provider is JBOSS.
- The server-side business logic consists of a Validator and Matcher, implemented as Message-Driven POJOs (MDP).
- The MDPs access a database using a Data Access Object (DAO).

This is the DAO interface, which remains constant throughout the transition:

```
public interface IOrderDao {
       public List findOrdersByVolume(int volume);
}
```

And this is the existing DAO implementation, which refers to a database, and uses Spring declarative transactions management (declared via annotation):

```
public class HibernateOrderDao implements IOrderDao {
       @Transactional
       public List findOrdersByVolume(int volume) {
              Session session = getHibernateSession();
              List orders = session.createQuery("from Order " +
                                   "where volume > ?")
                                                .setInteger(0, volume)
                                  .list();

              return orders;
       }
}
```

What Needs to Change?

To move the application to Step One, you first need to add the following beans to the Spring configuration, so you can wire the DAO to the IMDG instead of to the database:

```
<os-core:space id="space" url="/./test" lookup-groups="${user.name}"
    no-write-lease="true" />
<os-core:giga-space id="gigaSpace" space="space" />
<os-core:local-tx-manager id="localSpaceTransactionManager"
    giga-space="gigaSpace" />
```

Now all that's left is to switch the existing DAO implementation with the following. Note we are still using Spring declarative transactions management, which is supported by GigaSpaces.

```
public class GigaSpaceOrderDao implements IOrderDao {
       @Trasactional
       public List findOrdersByVolume(int volume) {
              SQLQuery query = new SQLQuery(Order.class,
                                           "where volume > ?");
              return Arrays.asList(gigaSpace.readMultiple(query));
       }
}
```

As you recall, persistence is completely decoupled from the application. You can add any of the persistence options discussed above via the UI of the IMDG, and change your persistence strategy at any time. The same goes for the underlying topology of the IMDG nodes, which may be partitioned, replicated, and so on.

2.5. Benefits: No I/O Bottleneck, Decoupled Persistence

Virtualizing the data immediately provides the following benefits:

- **I/O bottleneck is reduced**—There is no longer a need to write everything to disk, and this can dramatically reduce latency and boost performance.
- **Persistence is decoupled from the application**—Allows for simpler development and testing, and permits a flexible and dynamic persistence strategy.
- **Easier scaling of data**—The IMDG completely virtualizes the data. Additional nodes can be deployed on any available resources and can immediately join the existing topology.
- **No need to translate to relational model**—The IMDG is object-oriented, so there is no longer a need to translate data into a relational model and back again, further boosting performance.

2.6. Remaining Pain Points

Even after the data is virtualized, the following pain points remain, which are the primary motivation for moving on to **Step Two: Virtualizing Your Messaging**.

- **Large number of network hops**—Like in the classic Tier-based model, each transaction still has to go through a central messaging provider, limiting the reduction in latency.
- **Two-phase Commit still required**—When using the IMDG in its basic form,[7] state data needs to be guaranteed by a distributed transaction managed by a central transaction manager, which remains a bottleneck and a single point of failure.
- **Numerous moving parts**—The system is still complex, difficult to maintain, and susceptible to partial failure.
- And, as a result—still has **limited scalability.**

[7] As discussed in the following chapter, the IMDG can also perform the messaging function, which removes the need for a distributed transaction.

3. STEP TWO: Virtualizing Messaging

The contemporary Enterprise Service Bus (ESB) model was intended to enable workflow management without a central messaging component. In practice, however, the services-based approach is stateless, and very often the application needs to save state data. To accommodate this, most services end up writing their state to a database, which itself becomes a central dependency and a bottleneck. Furthermore, to guarantee affinity in case of partial failure, a central transaction manager is needed.

Put together, these factors actually make the ESB model less efficient than the old, central-server model. On the architectural level, ESB does not have the appeal of SOA/EDA because while the services may be decoupled from each other and from the application, they are still tightly coupled to the database and the transaction coordinator.

Consolidating the data and the messaging solves the problems of ESB, and permits stateful processing with guaranteed affinity (through the reliability and persistence strategies discussed in the previous chapter) with extremely low latency.

Transition: **Create a Data Bus that consolidates your messaging and your data.**

Result: **Messaging provider can be scaled without threatening affinity, and can be made highly available without writing to disk.**

3.1. Managing Data and Messages in the Same Place

The problem with the ESB is not the model, it's the need to rely on an external component—a database—to store state data. The solution is a *data bus*: a hub to which the business logic services connect, and that performs both the messaging function and the data storage function.

With the OpenSpaces IMDG, this is achieved with the ability to fire events when the data changes (this enables a publish-subscribe messaging model), and the ability to perform a blocking read request on data (which enables a queue model). This facilitates workflow management without an actual messaging provider. GigaSpaces provides a JMS façade which enables the application to use the familiar JMS API, so that the IMDG can easily replace an existing messaging provider without requiring any change to the application.

3.2. The Transition

Here is how the transition is achieved in our reference application from the previous chapter, which has already virtualized its data using the GigaSpaces IMDG.

The reference application's services use Spring's `jmsTemplate` to send JMS messages. This stays exactly the same. Behind the scenes, you wire the `jmsTemplate` bean to a Connection Factory and Destination provided by OpenSpaces:

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="defaultDestination" ref="destination" />
</bean>
<os-jms:queue id="destination" name="MyQueue" />
```

```
<os-jms:connection-factory id="connectionFactory" giga-space="gigaSpace" message-converter="messageConverter" />

<bean id="messageConverter"
class="com.j_spaces.jms.utils.ObjectMessage2ObjectConverter" />
```

The last bean is a message-to-object convertor, which will convert the JMS messages created by your services to objects that can be accepted by the GigaSpaces middleware components. This bean is wired into the Connection Factory, so message conversion is transparent to your services.

3.3. Benefits: Guaranteed Affinity, Fewer Moving Parts, Reduced Latency

Virtualizing the messaging tier immediately provides the following benefits:

- **Guaranteed affinity**—When data storage and messaging are performed by the same component, data affinity is guaranteed by definition, without needing to rely on a transaction coordinator.
- **Fewer moving parts**—This means fewer points of failure, simpler maintenance, and easier scalability.
- **Single clustering model**—When scaling out, the messaging and data are kept in the same cluster, reducing inefficiencies, administration headaches, and performance issues that arise due to unaddressed integration points between the clusters.

3.4. Remaining Pain Points

Even after consolidating the data with the messaging, several pain points remain, which are the primary motivation for moving on to **Step Three: Virtualizing Your Application**.

- **Network hops**—Although latency has improved, each transaction still needs to travel from the business logic tier to the data/messaging tier, and back again, several times.
- **Integration points**—There are still potentially problematic integration points between the data tier and the data/messaging tier.
- **Separate clustering models for business and data/messaging tier**—The business tier still needs to be scaled separately using a different clustering model from the data/messaging tier (which is now virtualized).
- **Non-trivial deployment**—Scaling out the application is a complex exercise requiring manual provision of separate resources for the business logic and data/messaging tiers.
- And, as a result— the system still has **limited scalability and suboptimal performance and latency.**

4. STEP THREE: Virtualizing Your Application

Even if you solve the individual bottlenecks caused by the data and messaging tiers, there is still something in the way of scalability—the business logic services are tightly coupled to the data and messaging. This is true for applications that do not practice virtualization, and remains true even if you virtualize the data and messaging, by performing Step One and Step Two above. The solution, as you'll discover below, is to package the business logic, the data and the messaging into a self-sufficient *Processing Unit*.

Transition: **Use the Spring Application Context to package your business logic together with the middleware, creating a *Processing Unit*.**

Result: **Application can be scaled linearly, by deploying the Processing Unit on as many machines as needed.**

4.1. Separation and Inter-Dependence of Tiers as a Bottleneck—A Factory Metaphor

To understand why the separation and inter-dependence of business logic and middleware is a bottleneck, consider the following story. Think of the business logic, data and messaging tiers as separate facilities that perform separate production functions in a Cola bottling plant. For example, filling soda bottles, capping them, and packaging them for the stores. Each of these three factories has its own building, its own machines, and its own management.

Now, the whole system needs to process twice as many bottles per day. The packaging factory can scale immediately—it just needs to recruit some more workers. But the bottle-capping factory needs to add a new production line, so it needs two months to scale up, and the bottle-filling factory needs to replace all its machines, so it will need six months to scale up. This means the whole system will take six months to deliver the required throughput—it is only as strong as its weakest link. The Tier-based Architecture wasn't designed to scale, and simply cannot scale on demand, because of its fundamental precept of separation and inter-dependency between tiers.

The solution? Instead of having each production function in a separate factory, we can build one factory with a single production line that performs the entire production process. It then becomes very easy to scale the operation by setting up additional, identical factories. These needn't be on the same site—they can be set up wherever resources are cheapest. When market conditions change, it is a simple matter of setting up more factories, or closing some of the factories, to achieve the required throughput, on-demand.

As the factory metaphor illustrates, the key to ramping up or reducing production volume is to create self-sufficient factories that can complete the entire workflow with no inter-dependencies. In much the same way, the scalability problems of Tier-Based applications can be solved by pacakaging all the tiers—business logic, messaging and data—into self-sufficient units that be cloned without limitation, and deployed on any available computing resources, when the application needs to scale out.

4.2. Achieving Self-Sufficiency

In a stateless environment, self-sufficiency is quite easy to achieve. But in a stateful environment, in which different components perform different parts of a structured workflow, a paradox arises: How can you manage a workflow when each machine is completely self-sufficient? How can machines share state information between them?

The key to resolving the paradox is *collocating* all steps of the business process—putting them on the same machine in the same VM, or Processing Unit. This way, information only needs to be shared **within the Processing Unit**—within the confines of the local machine—and each Processing Unit can manage its own workflow.

The Processing Unit does not have to be one monolithic unit—it can be composed of several subcomponents, which can even be defined as services participating in an EDA/SOA. The only requirement is that all these subcomponents be collocated in the same VM.

4.3. Packaging Business Logic, Data, and Messaging

GigaSpaces XAP allows developers to build business logic on a single machine and then deploy and scale it on a pool of machines. The infrastructure will take care of routing the appropriate request to the right service destination based on the request data or service availability.

A Processing Unit is defined using a Spring configuration file, which contains both the business logic services and a middleware component.[8] The middleware component is an IMDG node that provides not only data storage but also messaging functionality. This allows the Processing Unit to receive a request and perform the entire business process without relying on any external component, even middleware components, and return a result to the client.

Within the Processing Unit, the Polling and Notify Containers encase the business logic services and deliver events fired by the built-in middleware component.[9] These Containers, which are declared within the Spring Application Context in much the same way as a Message-Driven POJOs, process events delivered from the messaging and data components, and hold the state required for the business logic implementation.

To define the Processing Unit, all you need to do is wire the POJO, .NET or C++ object that represents your business service into the relevant Event Container, within the Spring Application Context.

---

[8] For more details, refer to *GigaSpaces Online Help*, OpenSpaces Processing Unit, http://www.gigaspaces.com/wiki/x/moCE.

[9] For more details, refer to *GigaSpaces Online Help*, OpenSpaces Event Containers, http://www.gigaspaces.com/wiki/x/bICE.

4.4. The Transition

Here is how the transition is achieved in our reference application from the previous chapter, which has already virtualized its data and messaging. First, the application drops the MDPs and instead, defines a Polling Container in its server-side Spring configuration:

```
<os-events:polling-container id="dataProcessorPollingEventContainer" giga-
space="gigaSpace">
    <os-events:tx-support tx-manager="transactionManager"/>
    <os-core:template>
        <!-- Business logic services -->
        <bean class="com.gigaspaces.scalingspring.validator>
            <property name="…" value="…"/>
            …
        </bean>
        <bean class="com.gigaspaces.scalingspring.matcher>
            <property name="…" value="…"/>
            …
        </bean>
    </os-core:template>
    <os-events:listener>
        <os-events:annotation-adapter>
            <!—- Objects in the domain model -->
            <os-events:delegate ref="order"/>
            …
        </os-events:annotation-adapter>
    </os-events:listener>
</os-events:polling-container>
```

As you can see, the Polling Container declares the object to listen on (the domain model, which hasn't changed in the transition), and the business logic services (the Validator and Matcher, in this case), which receive events from the OpenSpaces middleware component.

The Polling Container is defined in the broader context of a Processing Unit. The Processing Unit is just the full Spring Application Context, defining all the business logic services and other required beans. This is exactly the same as in a regular Spring application, except that all the services happen to be collocated on the same machine.

4.5. Benefits: Linear Scalability, Minimal Latency

Virtualizing and packaging the application's business logic with the virtual middleware stack provides the key benefits of the Space-based Architecture:

- **Linear scalability**—Each Processing Unit is completely self-sufficient and there are no central components.

- **Minimal possible latency**—The entire business process occurs at in-memory speed, with no network hops, guaranteeing minimal latency.

5. STEP FOUR: Virtualizing Deployment

The OpenSpaces Deployment Framework is responsible for abstracting the application from the underlying hardware resources it utilizes. It does this through an SLA-driven Container, which is a monitoring and management engine that automatically deploys application components, such as the data or business logic, to the most appropriate machines. The Container addresses spikes in business volume by dynamically scaling application resources, and ensures that service-level agreements are met by detecting and recovering from failures (i.e. self-healing).

Transition: **Deploy your application using an *SLA-driven container*.**

Result: **Proactive deployment of as many application instances as required, on any available computing resources, with instant self-healing in case of failure.**

5.1. SLA-driven Containers

OpenSpaces SLA-driven Containers (formerly known as Grid Service Containers) allow you to deploy your application as a Processing Unit over a dynamic pool of machines. A Processing Unit is a simple directory structure, compliant with the OSGi-Spring integration directory structure. It includes a Spring XML configuration file (under `META-INF/spring/pu.xml`), the business logic class files, and third-party module `jar` files. A Processing Unit, under this structure, can run locally within the IDE, and be deployed in the SLA-driven Container without any changes.

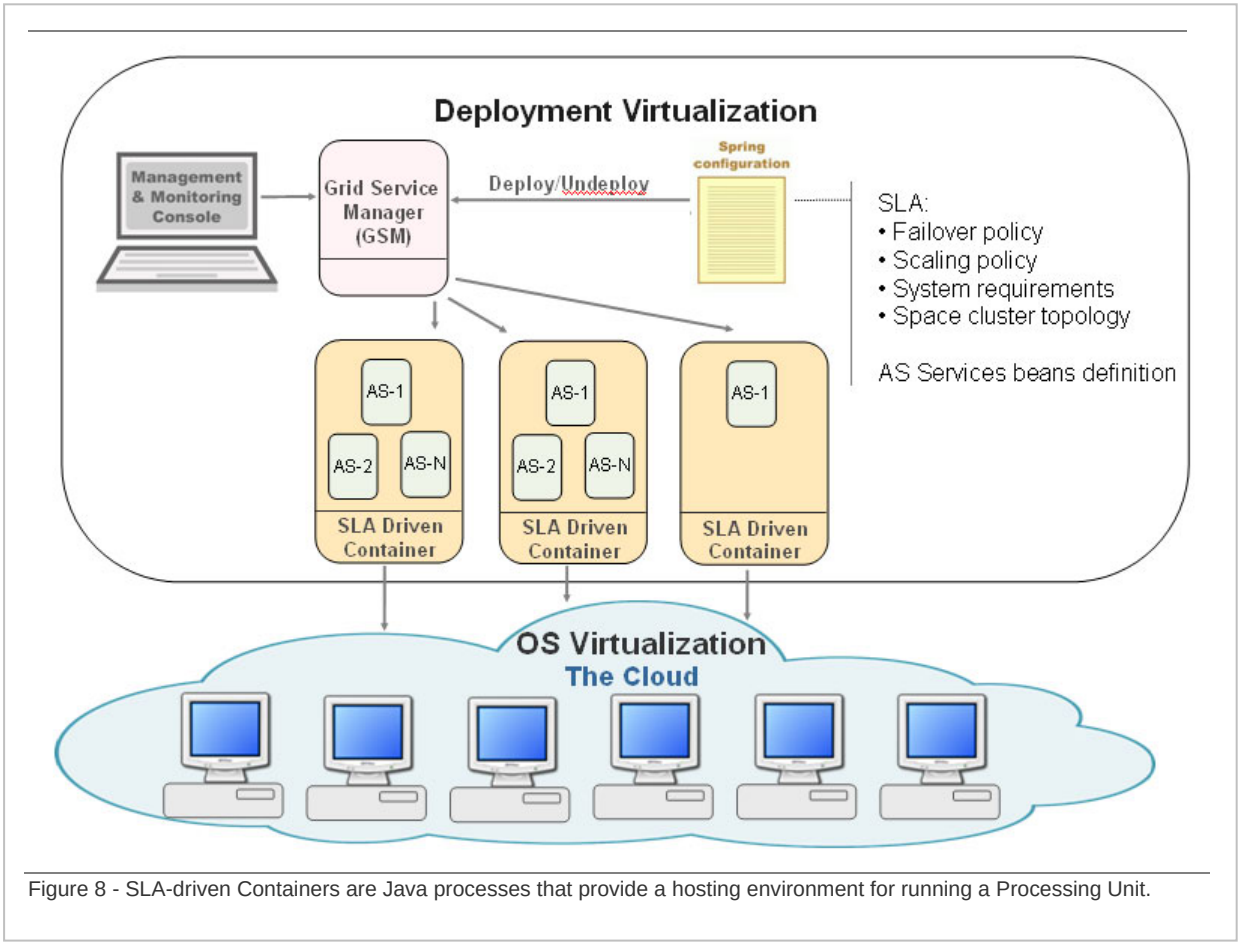


Figure 8 - SLA-driven Containers are Java processes that provide a hosting environment for running a Processing Unit.

A *deployment descriptor* defines which services each Processing Unit should comprise, their mutual dependencies, the software and hardware resources they require, and the SLA they are required to meet.

The *Grid Service Manager (GSM)* then goes into action, automatically deploying the Processing Unit on as many physical machines as necessary to meet the SLA.

The deployment framework performs **distributed dependency injection**—each time an instance of the Processing Unit is launched, the framework provides each service within the unit with a handle to all the other services it needs to work with. This means that services do not need to discover each other and check mutual availability. The deployment framework sees the entire deployment scope, and knows which services are currently available. A Processing Unit is **only deployed if all its dependencies can be currently fulfille**d.

The following is a snippet taken from the example SLA definition section of the Processing Unit Spring configuration:

```
<os-sla:sla cluster-schema="partitioned-sync2backup" number-of-instances="2"
number-of-backups="1"
            max-instances-per-vm="1">
    <os-sla:monitors>
        <os-sla:bean-property-monitor name="Processed Data"
                                      bean-ref="dataProcessedCounter"
                                      property-name="processedDataCount" />
    </os-sla:monitors>
</os-sla:sla>
```

5.2. Benefits: Single Model for Development and Deployment

Virtualizing deployment gives you:

- **Proactive deployment**—application instances are deployed just-in-time on the appropriate computing resources, and un-deployed when they are no longer required.
- **Provisioning sensitive to SLA and computing resources**—the deployment framework provisions as many instances as necessary to ensure that you meet your transaction load and your resiliency requirements (e.g. one or two backups per instance on separate machines).
- **Self-healing**—application instances which fail are instantly replaced by new instances on another machine, containing the same data, which immediately resume all operations from the precise point of failure. When the original instance is restored, data and operations are relocated back to it.
- **Aggregate monitoring**—one view of the entire deployment scope, with the ability to view the full path of a business transaction.

6. Welcome to SBA: Scalability Made Simple

If you choose to perform all four steps described in this paper, then without realizing it, you will have made a full transition to the Space-Based Architecture (SBA). The primary benefit of SBA is *simplicity*—the distributed application behaves as one coherent unit, as if it was a single server.

- **From the designer's perspective**—scalability is not a design consideration. The designer constructs a self-contained Processing Unit that can be duplicated as many times as needed to facilitate future scaling.
- **From the developer's perspective**—the system presents a unified API and clustering model. Developers continue to write code as if for a single server; the physical distribution of the runtime environment is completely abstracted through the middleware implementation and the Spring integration. Efforts can focus on a standalone Processing Uunit, in a standalone application context; this also enables testing of 90% of the business logic on a single machine, without running a cluster.
- **From the administrator's perspective**—the entire application is deployed through a single deployment command, as if it were running on a single server. The deployment command is SLA-driven, depending only on the predefined hardware and software requirements of each processing unit—the physical topology and machine availability is totally abstracted from the administrator.
- **From the client's perspective**—clients see the entire cluster of Processing Units as if they were a single server. They can subscribe to messages, perform queries, or even read data directly from the space of a Processing Unit in a single operation, without being aware of the underlying topology. Operations are maintained transparently by the SBA middleware.

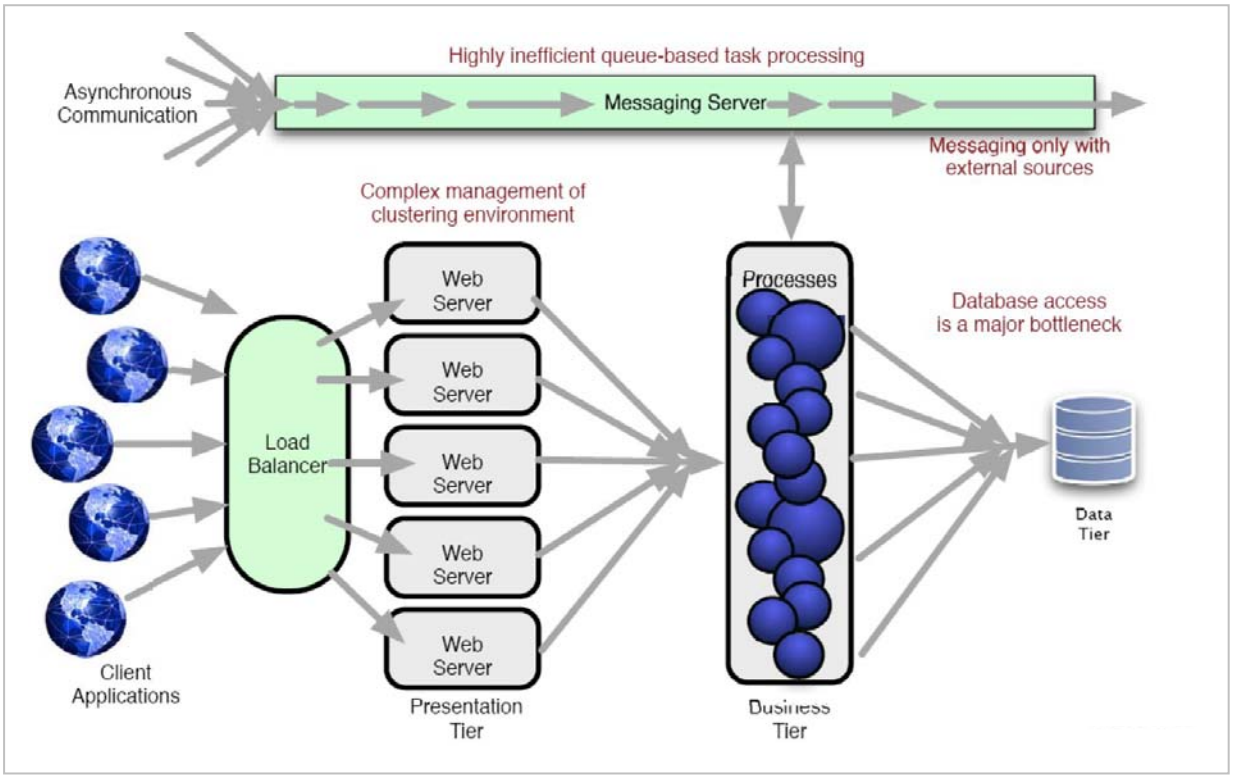
The bottom line is that **the application's scale becomes transparent** to all involved. Complexity, overhead, and cost—the three biggest challenges of a stateful application as it scales up—stay exactly the same as the distributed application grows and grows.

7. A Real-Life Example

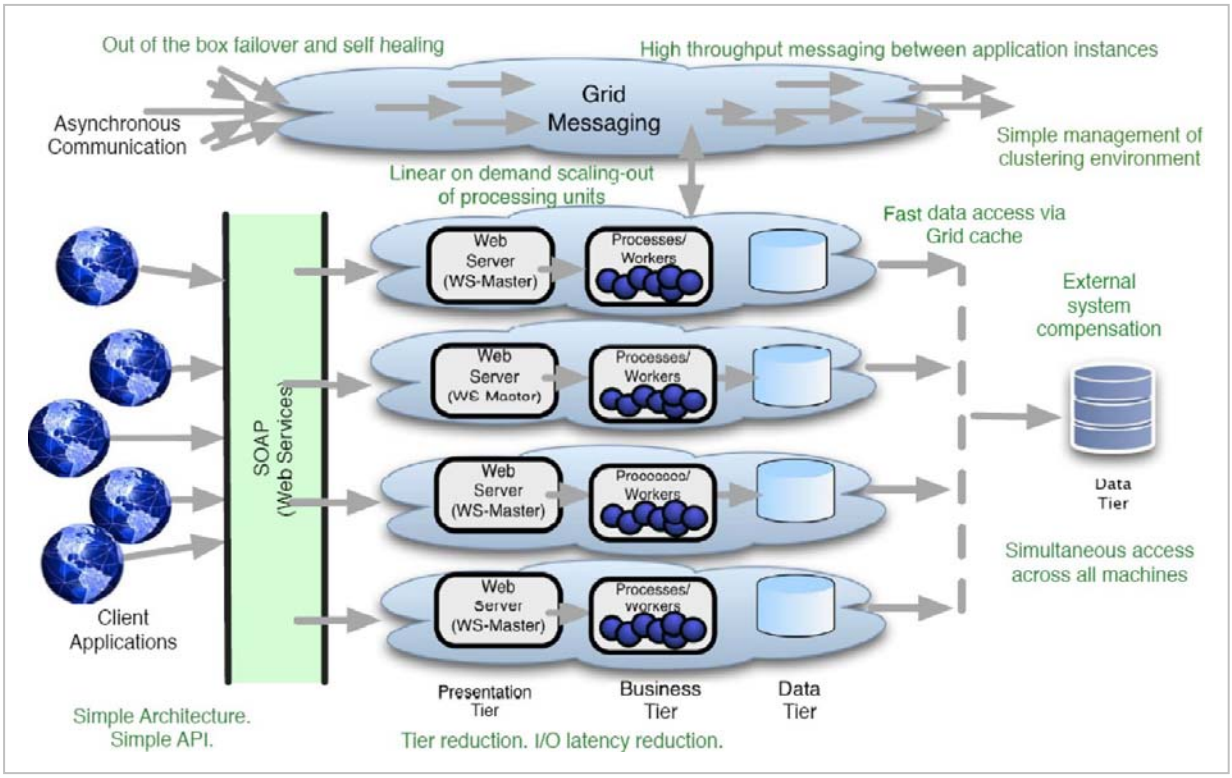
At the end of 2007 GigaSpaces engaged with a large wireless carrier that was about to launch a new campaign. The company was concerned about the scalability of their existing Tier-Based order management system due a recent failure in an earlier launch. They heard about Space-Based Architecture and wanted to use GigaSpaces to implement this pattern. A major hurdle was that most of the code was already written for a classic tier-based J2EE model and they had only four weeks to integrate with GigaSpaces and go directly into production.

GigaSpaces established a tiger-team that integrated GigaSpaces XAP with the customer's existing application, and within two weeks completed development and started stress testing. The system successfully went live before the holiday shopping season and served an extremely successful campaign. The diagrams below shows the before and after architecture of this particular application.

7.1. Real-Life Order Management System — Tier-Based Architecture



7.2. And After Four Weeks…



U.S. Headquarters
GigaSpaces Technologies Inc.
317 Madison Ave, Suite 1220
New York, NY 10017
Tel: 646-421-2830
Fax: 646-421-2859

U.S. West Coast Office
GigaSpaces Technologies Inc.
555 California Street, 3rd Floor
San Francisco, CA 94104
Tel: 415-568-2125
Fax: 415-651-8801

Europe Office
GigaSpaces Technologies Ltd.
2 Sheraton Street,
London, W1F 8BH, UK
Tel: +44-207-117-0213
Fax: +44-870-3835-135

International Office
GigaSpaces Technologies Ltd.
4 Maskit Street, P.O. Box 4063
Herzliya, 46140, Israel
Tel: +972-9-952-6751
Fax: +972-9-957-6780